```
; -------------------------------------------------------------------------------
; Progetto : Realizzazione di un risolutore e di un generatore di problemi di
;            CSP binario.
; Autori   : Alberto Schena            (#60735)
;            Nicola Fraccaroli         (#60733)
;            Giovanni Dall'Oglio Risso (#60734)
; Corso    : Intelligenza Artificiale A
; Docente  : Alfonso Gerevini
; -------------------------------------------------------------------------------

; -------------------------------------------------------------------------------
; MACRO
; -------------------------------------------------------------------------------

; NOME:
;   rnd
; DESCRIZIONE:
;   Genera un numero intero pseudo-casuale compreso nell'intervallo ['a' .. 'b']
;   (estremi inclusi). La distribuzione si approssima a quella di una variabile
;   uniforme discreta di parametri (a, b).
; PARAMETRI:
;   a (tipo: integer)
;     L'estremo inferiore (inclusivo) dell'intervallo.
;   b (tipo: integer)
;     L'estremo superiore (inclusivo) dell'intervallo.
; RITORNO:
;   (tipo: integer)
;     Il valore pseudo-casuale generato.
(defmacro rnd (a b) `(+ ,a (rand (1+ (- ,b ,a)))))
; -------------------------------------------------------------------------------

; NOME:
;   rep-until
; DESCRIZIONE:
;   Valuta 'form' finché 'pred' non risulta vero. Ritorna poi il valore corrente di
;   'form'.
; PARAMETRI:
;   pred (tipo: function; parametri: (qualsiasi); ritorno: boolean)
;     La funzione che riceve in ingresso il valore corrente di 'form' e restituisce
;     T se il ciclo di valutazione deve essere concluso, altrimenti restituisce NIL.
;   form (tipo: <qualsiasi>)
;     La forma il cui valore viene valutato ad ogni iterazione.
; RITORNO:
;   (tipo: <lo stesso di 'form'>)
;     Il valore di arresto del ciclo di valutazione.
(defmacro rep-until (#'pred form)
    `(do ((result ,form ,form)) ((funcall ,pred result) result))
)
; -------------------------------------------------------------------------------

; -------------------------------------------------------------------------------
; FUNZIONI
; -------------------------------------------------------------------------------

; NOME:
;   rand
; DESCRIZIONE:
;   Genera un numero intero pseudo-casuale compreso nell'intervallo [0 .. 'limit'-1].
```

```
;   La distribuzione si approssima a quella di una variabile uniforme discreta di
;   parametri (0, 'limit'-1).
;   Approssima meglio la distribuzione uniforme discreta rispetto all'implementazione
;   di random. Procedura suggerita da "Common Lisp: The Language, 2nd ed."
;   di Guy L. Steele Jr., pag. 392
; PARAMETRI:
;   limit (tipo: integer)
;     L'estremo superiore (esclusivo) dell'intervallo.
; RITORNO:
;   (tipo: integer)
;     Il valore pseudo-casuale generato.
(defun rand (limit)
    (mod (random (expt 2 (+ 20 (integer-length limit)))) limit)
)
; ------------------------------------------------------------------------------

; NOME:
;   choose
; DESCRIZIONE:
;   Restituisce un elemento scelto casualmente dalla lista 'lst'. Usa un generatore
;   di numeri casuali avente distribuzione prossima ad una variabile casuale uniforme
;   discreta: ogni elemento nella lista ha egual probabilità di essere selezionato,
;   pari a 1/LENGTH[lst].
; PARAMETRI:
;   lst (tipo: list)
;     La lista da utilizzare nella selezione.
; RITORNO:
;   (tipo: <dipende dall'elemento selezionato>)
;     L'elemento scelto casualmente dalla lista.
(defun choose (lst)
    (nth (rand (length lst)) lst)
)
; ------------------------------------------------------------------------------

; NOME:
;   shuffle
; DESCRIZIONE:
;   Permuta casualmente gli elementi nella lista 'lst' e restituisce la lista ottenuta.
;   Usa un generatore di numeri casuali avente distribuzione prossima ad una variabile
;   casuale uniforme discreta: ogni permutazione è in prima approssimazione
;   equiprobabile alle altre, pertanto ha probabilità di verificarsi pari
;   ad 1/LENGTH[lst]!.
; PARAMETRI:
;   lst (tipo: list)
;     La lista da permutare.
; RITORNO:
;   (tipo: list)
;     La lista permutata.
(defun shuffle (lst)
    (stable-sort (copy-list lst) #'(lambda (x y) (zerop (rand 2))))
)
; ------------------------------------------------------------------------------

; NOME:
;   csp-generate
; DESCRIZIONE:
;   Genera un modello di problema di CSP avente 'num-vars' variabili, ciascuna avente
;   per dominio un sottoinsieme dell'intervallo ['min-val'..'max-val']
```

```
;   (estremi inclusi). Ogni modello generato contiene un numero di vincoli almeno pari a
;   quello delle variabili ed ammette almeno una soluzione.
; PARAMETRI:
;   &key num-vars (tipo: integer; default: 10)
;     Il numero di variabili da inserire nel problema.
;   &key max-vals (tipo: integer; default: 10)
;     La cardinalità massima di ogni dominio di variabile.
;   &key min-val (tipo: integer; default: 0)
;     L'estremo inferiore (inclusivo) di ogni dominio di variabile.
;   &key max-val (tipo: integer; default: 100)
;     L'estremo superiore (inclusivo) di ogni dominio di variabile.
; RITORNO:
;   (tipo: csp)
;     Il modello di problema generato.
(defun csp-generate (&key
    (num-vars 10) (max-vals 10) (min-val 0) (max-val 100))

    (unless (>= num-vars 2)
        (error "Requisito: num-vars >= 2.")
    )
    (unless (>= max-vals 2)
        (error "Requisito: max-vals >= 2.")
    )
    (unless (>= max-val min-val)
        (error "Requisito: max-val >= min-val.")
    )
    (unless (>= (1+ (- max-val min-val)) num-vars)
        (error "Requisito: max-val - min-val + 1 >= num-vars.")
    )
    (unless (>= (1+ (- max-val min-val)) max-vals)
        (error "Requisito: max-val - min-val + 1 >= max-vals.")
    )
    (let ((vars nil)      ; Lista di nomi delle variabili.
          (refs nil)      ; Lista (di lunghezza num-vars) di indici ai valori in vals.
                          ; Più variabili possono puntare allo stesso valore.
          (vals nil)      ; Lista (di lunghezza casuale <= num-vars) di valori casuali
                          ; diversi.
          (equs nil)      ; Lista (di lunghezza pari a quella di vals) di indici delle
                          ; variabili aventi il valore corrispondente in vals.
          (constrs nil)   ; Lista dei vincoli.
          (domains nil)   ; Lista dei dominii.
          (pairs (make-list num-vars)))
        ; Inizializza la lista di variabili e dei rispettivi valori, creando
        ; vincoli di uguaglianza in numero tendente (in probabilità) a num-vars/4.
        (dotimes (i num-vars nil)
            (enqueue vars (list (make-symbol (format nil "var~a" (1+ i)))))
            (dotimes (j num-vars nil)
                (when (/= i j)
                    (enqueue (nth i pairs) (list j))
                )
            )
            (if (and (> i 0) (zerop (rand 4)))
                ; Probabilità 1/4: crea un'uguaglianza fra la variabile corrente
                ; ed una precedentemente creata. Imposta la i-esima variabile al
                ; valore j-esimo e crea un vincolo di uguaglianza fra la i-esima
                ; variabile ed una delle variabili associate al valore j-esimo
                ; scelta casualmente.
                ; NOTA: La coppia di variabili coinvolta dal vincolo è unica
```

```
            ;       per costruzione, pertanto non occorre il controllo con
            ;       is-necessary-constraint.
            (let* ((j (rand (length vals)))
                   (cmp (choose (nth j equs))))
                (enqueue refs (list j))
                (enqueue constrs
                    (if (zerop (rand 2))
                        (list (list '= (nth i vars) (nth cmp vars)))
                        (list (list '= (nth cmp vars) (nth i vars)))
                    )
                )
                (enqueue (nth j equs) (list i))
                (setf (nth i pairs) (remove cmp (nth i pairs)))
                (setf (nth cmp pairs) (remove i (nth cmp pairs)))
            )

            ; Probabilità 3/4: Imposta la i-esima variabile ad un nuovo valore
            ; diverso dai precedenti.
            (let ((j (length vals))
                  (val (rep-until #'(lambda (x) (not (member x vals)))
                    (rnd min-val max-val))))
                (enqueue refs (list j))
                (enqueue vals (list val))
                (enqueue equs (list (list i)))
            )
        )
    )

    ; Ispeziona nuovamente la lista di variabili e valori per generare i vincoli
    ; di disuguaglianza. Al termine di questo passo ogni variabile deve essere
    ; coinvolta in almeno un vincolo e non devono esistere vincoli che coinvolgono
    ; la stessa coppia (eventualmente invertita) di variabili.
    (dotimes (i num-vars nil)
        (when (nth i pairs)
            (let* ((j (choose (nth i pairs)))
                   (vi (nth (nth i refs) vals))
                   (vj (nth (nth j refs) vals))
                   (xi (nth i vars))
                   (xj (nth j vars))
                   (constr (list (get-relop vi vj) xi xj)))
                (enqueue constrs (list constr))
                (setf (nth i pairs) (remove j (nth i pairs)))
                (setf (nth j pairs) (remove i (nth j pairs)))
            )
        )
    )

    ; Crea i dominii delle variabili "nascondendo" il loro valore predeterminato
    ; in un insieme di valori casuali (distinti e compresi nell'intervallo
    ; ['min-val'..'max-val']). Al termine di questo passo ciascun dominio ha
    ; cardinalità casuale nell'intervallo [2..'max-vals']. Gli elementi di ogni
    ; dominio sono permutati casualmente in modo da rendere la risoluzione
    ; più difficoltosa.
    (dotimes (i num-vars nil)
        (enqueue domains
            (list
                (list
                    (nth i vars)
```

```
                        (shuffle
                            (remove-duplicates
                                (let* ((vi (nth (nth i refs) vals))
                                       (domain (list vi)))
                                    (dotimes (j (rnd 1 (- max-vals 1)) domain)
                                        (enqueue domain
                                            (list (rnd min-val max-val))
                                        )
                                    )
                                )
                            )
                        )
                    )
                )
            )
        )

        ; DEBUG ---
        ; (let ((result nil))
            ; (dotimes (i num-vars nil)
                ; (enqueue result
                    ; (list (list (nth i vars) (nth (nth i refs) vals)))))
            ; (princ (format nil "Result: ~%"))
            ; (print result)
        ; )
        ; DEBUG ---

        ; Crea il modello del problema di CSP utilizzando le liste costruite
        ; nei passi precedenti. La lista dei vincoli viene permutata casualmente.
        ; Il motivo sta nel fatto di voler "nascondere" i vincoli di uguaglianza
        ; che altrimenti si troverebbero nelle prime posizioni della lista dei
        ; vincoli. Ciò nega la possibilità di avere pattern di risoluzione
        ; ricorrenti.
        (make-csp
            :domains domains
            :constraints (shuffle constrs)
            :variables vars
        )
    )
)
; ------------------------------------------------------------------------------

; NOME:
;   get-relop
; DESCRIZIONE:
;   Restituisce un operatore di confronto scelto casualmente in modo da soddisfare la
;   relazione tra 'x' ed 'y'. I simboli possono essere: <, <=, >, >=, /=.
; PARAMETRI:
;   x (tipo: number)
;     Il primo termine nel confronto.
;   y (tipo: number)
;     Il secondo termine nel confronto.
; RITORNO:
;   (tipo: symbol)
;     Un simbolo di operatore di confronto.
(defun get-relop (x y)
    (cond
        ((= x y) (choose '(<= >=)))
```

```
        ((< x y) (choose '(< <= /=)))
        ((> x y) (choose '(> >= /=)))
    )
)
; -----------------------------------------------------------------------------

; NOME:
;   csp-save
; DESCRIZIONE:
;   Salva il modello 'csp' sui due file: FILE DEI DOMINII 'dom-file-path' e
;   FILE DEI VINCOLI 'con-file-path'. Restituisce il valore di 'csp' stesso.
;   Crea i file se non esistono; in caso contrario ne sovrascrive l'intero contenuto.
; PARAMETRI:
;   csp (tipo: csp)
;     Il modello da salvare.
;   dom-file-path (tipo: string)
;     Il percorso del FILE DEI DOMINII.
;   con-file-path (tipo: string)
;     Il percordo del FILE DEI VINCOLI.
; RITORNO:
;   (tipo: csp)
;     Il modello di problema specificato da 'csp'.
(defun csp-save (csp dom-file-path con-file-path)
    (with-open-file (dom-file dom-file-path :direction :output
        :if-exists :new-version :if-does-not-exist :create)
        (dolist (x (csp-domains csp) nil)
            (princ (cons (car x) (cadr x)) dom-file)
            (format dom-file "~%")
        )
    )
    (with-open-file (con-file con-file-path :direction :output
        :if-exists :new-version :if-does-not-exist :create)
        (dolist (x (csp-constraints csp) nil)
            (princ (list (cadr x) (car x) (caddr x)) con-file)
            (format con-file "~%")
        )
    )
    csp
)
; -----------------------------------------------------------------------------

; NOME:
;   massive-test
; DESCRIZIONE:
;   Esegue in sequenza 'num-tests' generazioni e risoluzioni di problemi di CSP.
;   Restituisce il numero di problemi INFEASIBLE. Tale contatore dovrebbe valere zero,
;   dato che il generatore di problemi parte da una soluzione predeterminata (quindi
;   ogni problema deve ammettere per ipotesi almeno una soluzione).
; PARAMETRI:
;   &optional num-vars (tipo: integer; default: 20)
;     Il numero di variabili di ciascun problema generato.
;   &optional num-tests (tipo: integer; default: 1000)
;     Il numero di test da generare.
; RITORNO:
;   (tipo: integer)
;     Il numero di problemi INFEASIBLE.
(defun massive-test (&optional (num-vars 20) (num-tests 1000))
    (let ((infeasibles 0)
```

```
          (mv (* num-vars 2)))
        (dotimes (i num-tests infeasibles)
            (let ((prob (csp-generate
                    :num-vars num-vars
                    :max-vals mv
                    :min-val 0
                    :max-val mv)))
                (if (symbolp (csp-solve prob :verbose nil))
                    (prog1
                        (incf infeasibles)
                        (princ (format nil "~a: ERRORE~%" i))
                    )
                    (prog1
                        (princ (format nil "~a~%" i))
                    )
                )
            )
        )
    )
)
; -------------------------------------------------------------------------------
```